Using Execution Data from Programs for Digital Forensics

Divya Paikaray, Assistant Professor

Department of Computer Science, Arka Jain University, Jamshedpur, Jharkhand, India

Email Id-divya.p@arkajainuniversity.ac.in

ABSTRACT: Computers and software are used by criminals to carry out their crimes or to hide their wrongdoings. The main memory, often known as RAM, contains a wealth of information about a system, including its running activities. The scope and duration of the variables data and value in RAM varies. To acquire proof of software use, this paper takes advantage of the program's execution state and dataflow. It collects data left behind from program execution in order to assist legal action against the offenders. Our research approach assumes that the operating system provides no information other than raw RAM dumps. The information we need comes from the source code of the target application. This paper focuses on C programs that run on Unix platforms. Several tests are carried out to demonstrate that the scope and storage information of different source code variables may be utilized to determine the program's actions. The findings indicate that even after the process has been halted, investigators have a high probability of finding the values of different variables.

KEYWORDS: Digital Forensics, Dumps, Memory, Memory Forensics, RAM.

1. INTRODUCTION

Computers and software are used by criminals to carry out their crimes or to hide their wrongdoings [1], [2]. Locating a program on the hard drive of a computer may not be sufficient to determine its specific use. To prove that the offender really utilized the software, proof may be required. This proof may be discovered in a few locations, one of which is the RAM of the computer in question. This highlights the importance of memory forensics and its use in criminal investigations.

Programs differ in their reliance on memory, CPU, disk I/Os, and networks in general. The control flow of a program may be heavily reliant on numerous variables and their values stored in several primary memory regions (RAM) [3]. The scopes and execution lives of these variables may be classified. The visibility of a variable and where it may be accessed inside the source code of a program is determined by its scope. The storage (memory type) of a variable, on the other hand, controls how long its value is generated, destroyed, or erased. Variables may also be categorized depending on whether or not they can be modified during execution. Constant variables are ones that can't be altered after they've been assigned, and they're usually given literal values (hard coded values). These literals may be specific to the executable program and its current stage of execution. Moreover, many additional non-constant variables may be set to hard coded values (literals).

Assume no information from the operating system is accessible; only raw RAM dumps are available [4]. This study looks for evidence that may be utilized to prove the software's use and link to the crime. The scopes of variables and memory kinds are the foundations of our inquiry paradigm. Various experiments and situations are created to validate our study approach. Based on the program source code and execution state, RAM memory dumps are generated and examined to find relevant variables' values (literal and non-literal).

This paper focuses on Unix-based systems and C applications. The majority of our results, however, are relevant to other languages and operating systems as well. Our findings demonstrate that regardless of whether the process is running or not, the memory investigator may ensure program use by using knowledge of the program source code and variables such as global and local static and their possible values. As a result, while the relevant stack frames are still active, the values of local auto variables are correctly found. Dynamically allocated values, on the other hand, may be found as long as the application is not terminated and the associated memory is not freed.

The remainder of this work is structured in the following manner. Section 2 summarizes some of the prior research and provides a literature review. Section 3 explains our investigation methodology and how it uses information from the program source code to verify that the application is in fact being utilized. Section 4 highlights our

encouraging findings after presenting our four trials. Finally, Section 5 summarizes our results and outlines our plans for future study.

2. LITRATURE REVIEW

Many experts believe that the RAM memory contains valuable data that may be utilized to support legal proceedings against offenders in digital forensic investigations. ForenScope is a RAM forensic tool developed by Chan Ellick et al. that allows users to examine a system using a normal bash-shell [5]. It enables users to turn off anti-forensic software and look for possible evidence. It is intended to operate in the unused memory space on the target computer in order to keep the RAM memory untouched.

Ahmad Shosha et al. created a prototype to identify a variety of harmful applications often employed by thieves [6]. The suggested method is based on the derivation of evidences based on traces associated with the suspicious software. Arasteh et al. gather evidence from RAM memory based on the process logic derived from its stack memory section [7].

FATKit was created by Petroni et al. It's a digital forensic tool for extracting, analyzing, and visualizing forensic data [8]. During memory dump analysis, it makes use of program source code and its data structure. To extract user input information from Windows programs, Funminiyi Olajide et al. utilize RAM dumps [9].

The prefetch folder and its possible usefulness to the investigator were the focus of N. Shashidhar et al [10]. On a Windows machine, this prefetch folder is used to speed up the starting time of an application.

Research Question:

1. What is the need of Execution Data in Digital Forensics?

3. METHODOLOGY

3.1. Design:

The visibility of a variable inside the source code of a program is affected by its scope, and the lifetime of a variable is affected by its storage. The survival of a variable's value in memory throughout multiple execution states may be affected by different scopes and storages. As a result, finding these numbers in a RAM memory dump may be used as proof that the assumed application was really utilized by a user. Our research design looks at the feasibility of finding the values of these variables across various scopes and storage kinds. Three distinct scopes are investigated in our experiments: global, local auto, and local static. As illustrated in Figure 1, it also attempts to differentiate between distinct values inside different execution states.

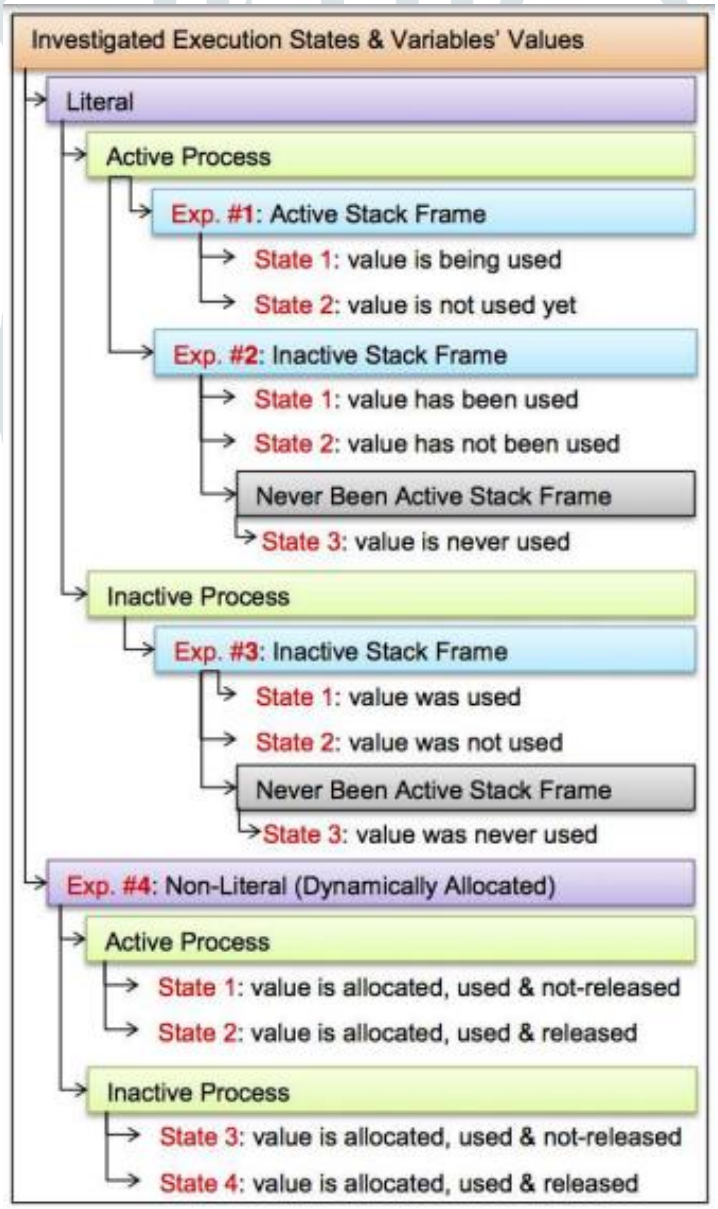

Figure 1: Various states of variables that are investigated throughout our studies. #1 represents a literal value within an active frame and a live process. #2 represents a literal value within a currently inactive frame and a live process. #3 represents a literal value within a currently inactive frame and a dead process. #4 represents a dynamically allocated value first within a live process and then within a dead process.

These execution states are based on a variety of situations, including the following:

- The variable was utilized in a stack frame that is presently active or inactive.
- During program execution, the variable is utilized or not yet used.
- The data of the assigned variable is never or only disclosed.
- The variable is never utilized; it is never accessed; and the stack frame is never active.
- The software process is either active or inactive.

In addition, our research model assumes that the operating system provides no information other than memory dumps generated throughout different execution stages and situations. Then, as illustrated in Figure 2, each memory dump is examined for possible values linked to the assumed program's source code.

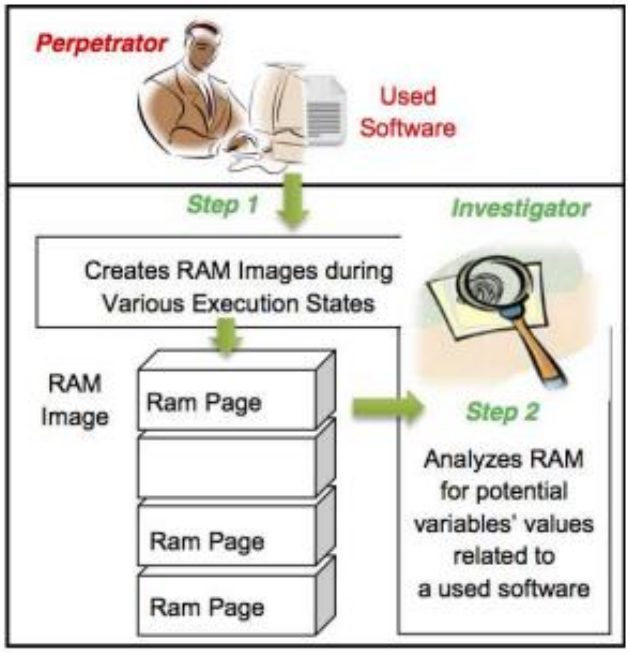

Figure 2: Investigation Model. The method of generating a memory dump is shown in Step 1. Step 2 is the process of looking for possible values in the source code of the target application.

3.2. Instrument:

We utilized a Linux virtual computer built using VirtualBox in all four tests. The virtual machine runs openSUSE Linux 13 and has 512 MB of RAM. This virtual machine is running on a Mac OS X 10. Oracle Corporation created VirtualBox, a free and open-source hosted hypervisor for x86 virtualization. It was created by Innotek and purchased by Sun Microsystems in 2008, which was then purchased by Oracle in 2010. VirtualBox is compatible with Windows, Mac OS X, Linux, Solaris, and OpenSolaris.

3.3. Data Collection:

For data collection a set of four experiments were conducted. Each of the four studies is intended to look for data that would show real program use throughout different execution stages. A literal or non-literal value may be assigned to a variable. Non-literal values are those that are computed, changed, and assigned dynamically during the execution of a program. As a result, experiments 1, 2, and 3 are intended to look at variables associated with string literals in various execution stages. Experiment 4, on the other hand, is intended to look at variable values that are dynamically allocated and changed. The scopes of three distinct variables are investigated in our experiments: global, local auto, and local static.

3.3.1. Experiment 1:

The purpose of the first experiment is to see whether using a literal string in a presently active stack frame impacts the ability to find that text in a memory dump generated during a live process. It looks at three distinct variable scopes: global, local auto, and local static, all of which are initialized with a string literal at declaration time. Within an active stack frame of an active process, it investigates two distinct states:

- State 1: The variable is utilized and is accessed in one of the statements that has been performed.
- State 2: The variable has yet to be utilized and it has not been accessed in any of the statements that have been performed so far.

For each of the investigated variables, a memory dump is captured in each of these states. The topic variable and its literal string value are searched in each of these dumps. Section 4 presents the conclusions of our investigation.

3.3.2. Experiment 2:

The purpose of the second experiment is to see whether using a literal string in an inactive stack frame impacts the ability to find that text in a memory dump. This experiment, like the previous, targets live processes using three distinct variable scopes: global, local auto, and local static, including one that is initialized with a string

literal at declaration time. Within an idle stack frame of an active process, it investigates three distinct variable states:

- State 1: The variable was utilized in one of the executed statements and it was read or assigned.
- State 2: The variable was not utilized in any of the previously performed statements and it was not read or assigned.
- State 3: The stack frame has never been active, and the function has never been invoked.

For each of the three investigated variable scopes, a memory dump is captured in each of these states. The literal values are then looked for in these dumps. Section 4 discusses the results of our investigation.

3.3.3. Experiment 3:

The third experiment is quite similar to the previous two. Except it looks at the impact of having an inactive process (one that has been halted) on the same three scopes and three states as the second experiment. For each of the three investigated variables' scopes, a memory dump is captured in each of these situations. The literal values are then looked for in these dumps. Section 4 presents the conclusions of our investigation.

3.3.4. Experiment 4:

The fourth experiment is intended to look at dynamically allocated string variables and compare them to variables in the source code that are initialized with literal values. The purpose of this experiment is to see whether it is possible to find variables' values that have been allocated in heap memory.

Variables are loaded into memory using malloc() as well as allotted a string value from another string literal using the strcpy() function in this experiment, after which some characters are modified to distinguish the string that dwells in the automatically assigned heap space from the original literal string. It investigates the possibility of finding these string values in four states:

- malloc(), strcpy(), and one character are changed in state one.
- State 2: malloc(), strcpy(), one character is changed, then free() is invoked.
- State 3: malloc(), strcpy(), one character is changed, free() is called, and the process is ended properly.
- State 4: malloc(), strcpy(), one character is changed, and afterwards trigger SIGINT is used to abnormally stop the program. The user application does not use free() directly.

States 3 and 4 are particularly intended to investigate the effects of having a process that terminates properly vs a process that terminates unexpectedly. In each of these situations, a memory dump is captured. The dynamically created and changed string values are then looked for in these memory dumps. Section 4 discusses the results of our investigation.

4. RESULTS AND DISCUSSION

This section summarizes the findings from all four experiments. The values of global and local static variables have two occurrences each, according to the results of the first experiment. The value of the local auto variable, on the other hand, appears just once in both states (State 1 and State 2), as shown in Table 1. This implies that whether the referenced variable is utilized or not in an active stack frame has no effect on the number of instances of the searched value that may be discovered. It also implies that the investigator may discover two instances of global and local static variables that are both initialized with literal strings, but only one instance of local auto variables.

Table 1: Illustrates that the global and local static variables have two occurrences each, while the value of the local auto variable has just one occurrence. States 1 and 2 demonstrate that whether the variable is utilized or not has no effect on the number of occurrences in all scopes studied.

Var Scope	State 1	State 2
Global	2	2
Local (Auto)	1	1
Local (Static)	2	2

The values of global and local static variables are discovered twice in the RAM dump, according to the results of the second experiment (two occurrences). The value of the local auto variable, on the other hand, is never discovered in any of the three states examined. As a result, having an idle stack frame during a live operation decreases our chances of finding the values of local auto variables. In our research setting, which comprises of relatively modest applications, having an active or inactive stack frame has no effect on the values of global and local static variables. Table 2 summarizes our results for each of the three variables' scopes as well as each of the three examined states.

Table 2: Illustrates that the global and local static variables possess two instances each, while the value of the local auto variable includes zero instances. States 1, 2, and 3 demonstrate that using or not using the variable has no effect on the number of occurrences in any of the examined states, even though the stack frame has never been active.

Var Scope	State 1	State 2	State 3
Global	2	2	2
Local (Auto)	0	0	0
Local (Static)	2	2	2

The third experiment's findings indicate that perhaps the value of the local auto variable has never been discovered in any of the three execution states examined (zero occurrence). This is consistent with the findings of the second experiment. The frequency distribution of values of global and local static variables, on the other hand, is reduced from two to one for each variable in each state. This implies that the investigator only gets one opportunity to find literal values of global and local static variables if the process is dormant (dead) (one occurrence). Table 3 summarizes our results for each of the three scopes and the three states examined.

Table 3: Illustrates that the global and local static variables have just one occurrence in the RAM dump, while the local auto variable has zero occurrence in the RAM dump. This is true in all three states, which implies that whether the variable is utilized or not has no effect on the outcomes, but whether the process is active or inactive does. States 1, 2, and 3 demonstrate that using or not using the variable has no effect on the number of occurrences in any of the examined states, even though the stack frame has never been active.

Var Scope	State 1	State 2	State 3
Global	1	1	1
Local (Auto)	0	0	0
Local (Static)	1	1	1

The results of the fourth experiment show that the investigator has an opportunity to locate an automatically assigned string value that resides in heap memory for an automatically assigned variable as long as the process is active and the value is not explicitly released in the program (the free() function is not called). Otherwise, we

have no possibility of finding any of these string values; at least, not in our test environment. The frequency distribution for the studied string value in four distinct execution stages is shown in Table 4.

Table 4: In State 1 i.e., when the process is active but the free() function is not invoked, the results of the fourth experiment indicate that global, local auto, and local static variables all have just one occurrence. In the other three stages, however, all variables' scopes have zero occurrences. Only in State 1 do we have a chance to identify dynamically allocated strings.

Var Scope	State 1	State 2	State 3	State 4
Global	1	0	0	0
Local (Auto)	1	0	0	0
Local (Static)	1	0	0	0

5. CONCLUSION AND IMPLICATION

This paper uses information from a program's source code as well as data from the program's execution during different execution states to aid investigators in establishing evidence against a perpetrator. Law enforcement will be able to pursue legal action against offenders in a court of law as a result of this. The C programming language is used in our research. We discovered that using source code knowledge may be beneficial to the investigator based on these studies. It aids in establishing proof that the offender used the program to commit the crime or cover up the violation. During different situations and execution states, numerous string literals and non-literals relevant to program execution are successfully found.

We want to explore additional settings in the future, including Windows, Mac, and tiny devices such as phones and tablets. Some languages have their own memory management system and virtual machine, while others, such as C, rely on the operating system to manage their memory. We want to look at the differences in behaviour across programming languages like C++, Java, C#, and Python. We're also excited to explore into comparable situations involving different data kinds and data structures. Finally, it's critical to look at different kinds and their effects on long-running programs like servers.

REFERENCES

[1] H. M. A. Van Beek, E. J. Van Eijk, R. B. Van Baar, M. Ugen, J. N. C. Bodde, and A. J. Siemelink, "Digital forensics as a service: Game on," *Digit. Investig.*, 2015, doi: 10.1016/j.diin.2015.07.004.

[2] M. D. Kohn, M. M. Eloff, and J. H. P. Eloff, "Integrated digital forensic process model," *Comput. Secur.*, 2013, doi: 10.1016/j.cose.2013.05.001.

[3] N. M. Karie and H. S. Venter, "Taxonomy of Challenges for Digital Forensics," *J. Forensic Sci.*, 2015, doi: 10.1111/1556-4029.12809.

[4] X. Du, N. A. Le-Khac, and M. Scanlon, "Evaluation of digital forensic process models with respect to digital forensics as a service," 2017.

[5] E. Chan, W. Wan, A. Chaugule, and R. Campbell, "A Framework for Volatile Memory Forensics," *ACM Conf. Comput. Commun. Secur.*, 2009.

[6] A. F. Shosha, L. Tobin, and P. Gladyshev, "Digital forensic reconstruction of a program action," 2013, doi: 10.1109/SPW.2013.17.

[7] A. R. Arasteh and M. Debbabi, "Forensic memory analysis: From stack and code to execution history," *Digit. Investig.*, 2007, doi: 10.1016/j.diin.2007.06.010.

[8] N. L. Petroni, Aa. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digit. Investig.*, 2006, doi: 10.1016/j.diin.2006.10.001.

[9] F. Olajide, N. Savage, G. Akmayeva, and C. Shoniregun, "Identifying and Finding Forensic Evidence From Windows Application," *J. Internet Technol. Secur. Trans.*, 2012, doi: 10.20533/jitst.2046.3723.2012.0016.

[10] N. Shashidhar and D. Novak, "Digital Forensic Analysis on Prefetch Files.," *Int. J. Inf. Secur. Sci.*, 2015.